But first....

- One more pattern to make your lives simpler

The King of Compound Patterns

If Elvis were a compound pattern, his name would be Model-View-Controller, and he'd be singing a little song like this….

World Wide Developers Conference (2005)
Model, View, Controller
Lyrics and music by James Dempsey (Apple Operations Engineer)

Model View, Model View, Model View Controller
MVC's the paradigm for factoring your code,
into functional segments so your brain does not explode.
To achieve reusability you gotta keep those boundaries clean,
Model on the one side, View on the other, the Controller's in between.

Model View — It's got three layers like Oreos do.
Model View creamy Controller

Model objects represent your applications raison d'être.
Custom classes that contain data logic and et cetra.
You create custom classes in your app's problem domain,
then you can choose to reuse them with all the views,
but the model objects stay the same.

4

You can model a throttle in a manifold,
Model level two year old.
Model a bottle of fine Chardonnay.
Model all the twaddle stuff people say.
Model the coddle in a boiling eggs.
Model the waddle in Hexley's legs.

One, two, three, four.
Model View — You can model all the models that pose for GQ.
Model View Controller

View objects tend to be controls that view and edit,
Cocoa's got a lot of those, well written to its credit.
Take an NSTextView, hand it any old Unicode string,
the user interacts with it, it can hold most anything.
But the view don't knows about the Model:
That string could be a phone number or the words of Aristotle.
Keep the coupling loose and so achieve a massive level of reuse.

5

Model View — All rendered very nicely in Aqua blue
Model View Controller

You're probably wondering now.
You're probably wondering how,
the data flows between Model and View.
The Controller has to mediate,
between each layer's changing state,
to synchronize the data of the two.
It pulls and pushes every changed value.
Yeah.

Model View — mad props to the smalltalk crew!
for Model View Controller

Model View — it's pronouced Oh Oh not Uh Uh
Model View Controller

6

There's a bit more on this story, a few more miles upon this road,
well nobody seems to get much glory writing controller code.
Well the model is mission critical and gorgeous is the view,
But I'm not being lazy, but sometimes it's just crazy
how much code i write is just glue.
And it wouldn't be so tragic,
but the code ain't doing magic: it's just moving values through.
And I wish I had a dime for every single time
I set a TextField's stringValue.

Model View — how we're gonna deep—six all that glue
Model View Controller

7

Controller's know the Model and View very
uahh — intimately
They often are hardcoding which is very verboten for reusability.
But now you can connect any value you select to any view property.
And I think you'll start binding,
then you'll be finding less code in your source tree.
Yeah I know I was astounded,
that's not even a rhyme.

But I think it bares repeating
all the code you won't be needing,
when you hook it up in IB.

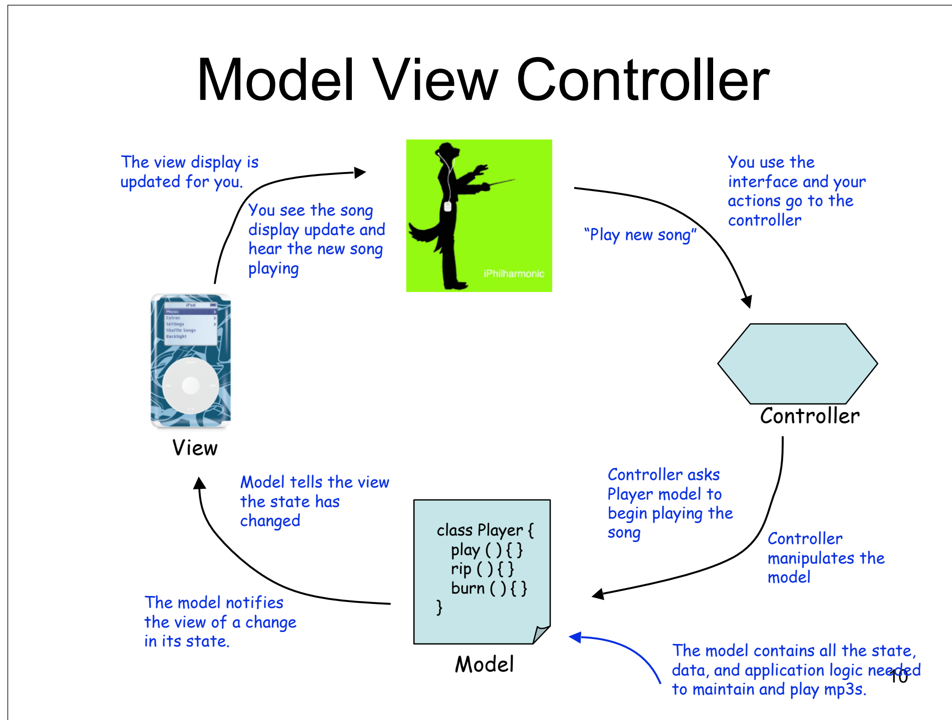
Model View — it even handles multiple selections too
Model View Controller
Model View — hope I get my G5 before you
Model View Controller
Yeah, yeah, yeah. Yeah.

8

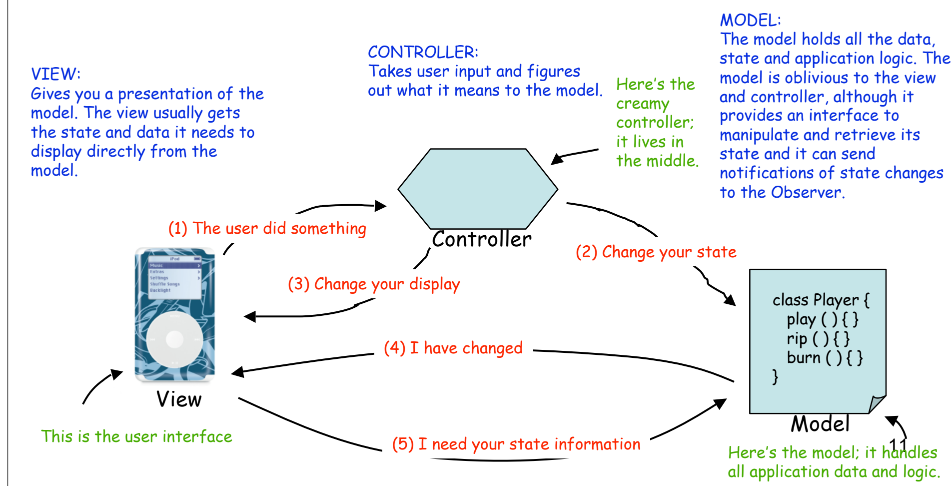
Meet the Model-View-Controller

- Picture your favorite MP3 player (iTunes, ?)
  - You can use its interface to add new songs, manage play lists and rename tracks
  - The player maintains
    - Database of all your songs along with associated names and data.
    - Plays the song
    - Updates the interface constantly with current song, running time and so on
- Underneath the covers --- Model View Controller

9



Model View Controller
The view display is updated for you.
You see the song display update and hear the new song playing
iPhilharmonic
"Play new song"
You use the interface and your actions go to the controller
View
Controller
Model tells the view the state has changed
Controller asks Player model to begin playing the song
Controller manipulates the model
class Player {
play ( ) { }
rip ( ) { }
burn ( ) { }
}
The model notifies the view of a change in its state.
Model
The model contains all the state, data, and application logic needed to maintain and play mp3s.
10



A Closer Look….
• Lets see the nitty gritty details of how this MVC pattern works….
– Step through the relationships among the model
– Take a look from Design pattern perspective.
VIEW:
Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.
CONTROLLER:
Takes user input and figures out what it means to the model.
Here's the creamy controller; it lives in the middle.
MODEL:
The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to the Observer.
(1) The user did something
Controller
(2) Change your state
(3) Change your display
(4) I have changed
class Player {
play ( ) { }
rip ( ) { }
burn ( ) { }
}
View
(5) I need your state information
Model
This is the user interface
Here's the model; it handles all application data and logic.
11

A Closer Closer Look….

(1) You are the user -- you interact with the view

– The view is your window to the model. When you do something to the view (like click the Play button), then the view tells the controller what you did. It's the controller's job to handle that.

(2) The controller asks the model to change its state

– The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.

(3) The controller may also ask the view to change

– When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.

(4) The model notifies the view when its state has changed.

– When something changes in the model, based on either some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.

(5) The view asks the model for state

– The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. The view might also ask the model for state as the result of the controller requesting some change in the view.

12

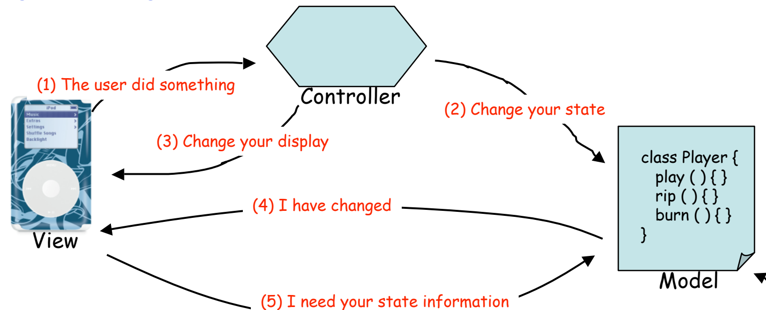
Looking at MVC through patterns-colored glasses

- MVC is set of patterns together in the same design.
- *Model* uses Observer to keep views and controllers updated on the latest state changes.
- *View* and *Controller* implement a Strategy Pattern
  - Example: Controller is the behavior of the view and can be easily exchanged with another controller if you want different behavior.
- *View* also uses a pattern internally to manage the windows buttons and other components of the display => the Composite Pattern

13

A Closer Look….

Strategy:
The View and the Controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy pattern also keeps the view decoupled from the model, because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how it gets done.



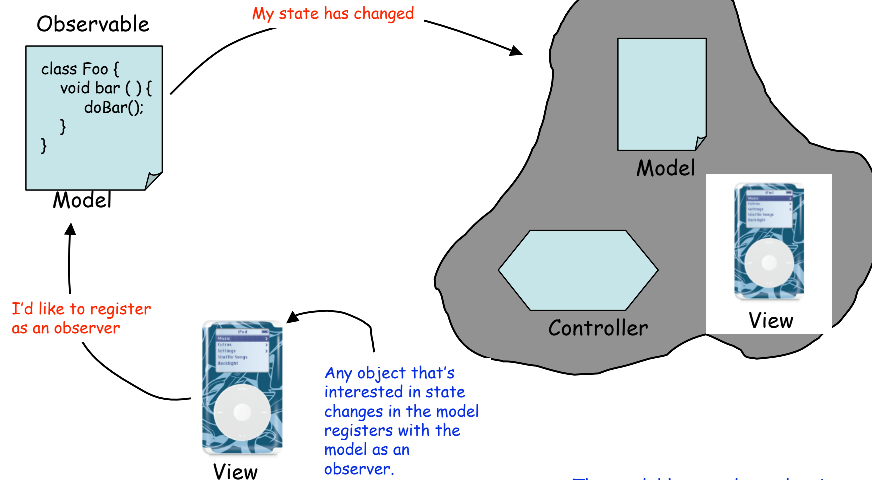
The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (button). When the controller tells the view to update, it only has to tell the top view component, and the Composite takes care of the rest.

The model implements the Observer Pattern to keep the interested objects updated when the state changes occur. Using the Observer Pattern keeps the model completely independent of the views and the controllers. Its allows us to use different views with the same model, or even multiple views at once.

14

Observer

All these observers will be notified whenever state changes in the model.



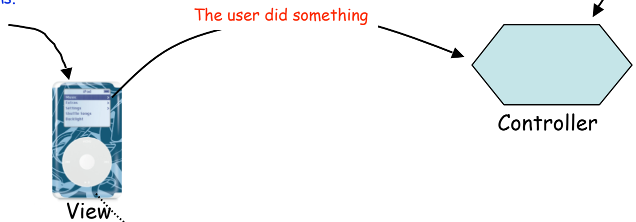
Any object that's interested in state changes in the model registers with the model as an observer.

The model has no dependencies on viewers or controllers!

Strategy

The view delegates to the controller to handle the user actions.

The controller is the strategy for the view -- it's the object that knows how to handle the user actions.

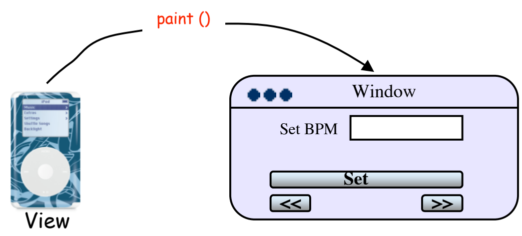


We can swap another behavior for the view by changing the controller.

The view only worries about presentation, the controller worries about translating user input to actions on the model.

16

Composite

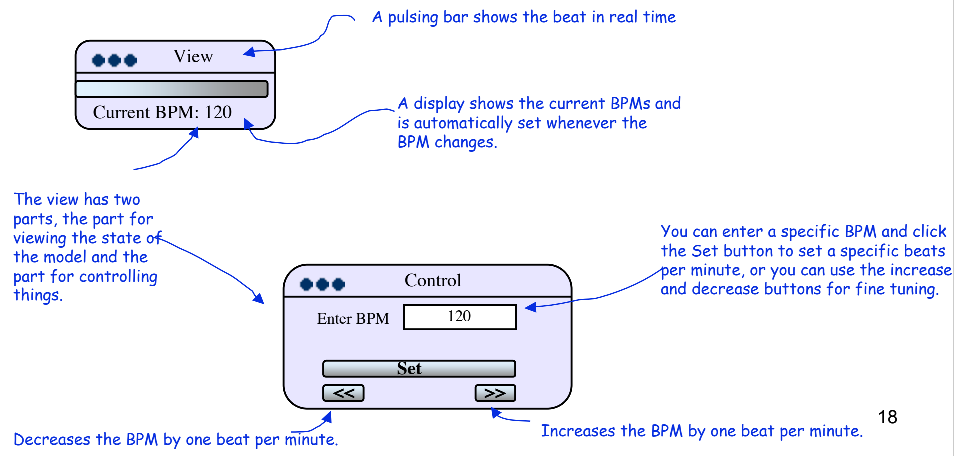


The view is a composite of GUI components (labels, buttons, text entry, etc.). The top level component contains other components, which contain other components, and so on until you get to the leaf nodes.

17

Using MVC to control the beat…

- Time to be a DJ and start mixing with a beat!
- The Java DJ view:



18

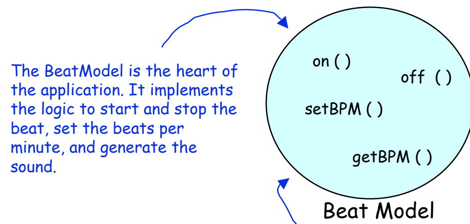
The Controller is in the middle…

- The controller sits between the view and the model. It takes your input, like decreasing the BPM and turns it into an action on the model to decrease the BPM.

19

Lets not forget the model underneath

The BeatModel is the heart of the application. It implements the logic to start and stop the beat, set the beats per minute, and generate the sound.



The model also allows us to obtains its current state through the getBPM( ) method.

20

Building the pieces

This gets called after the BeatModel is instantiated.

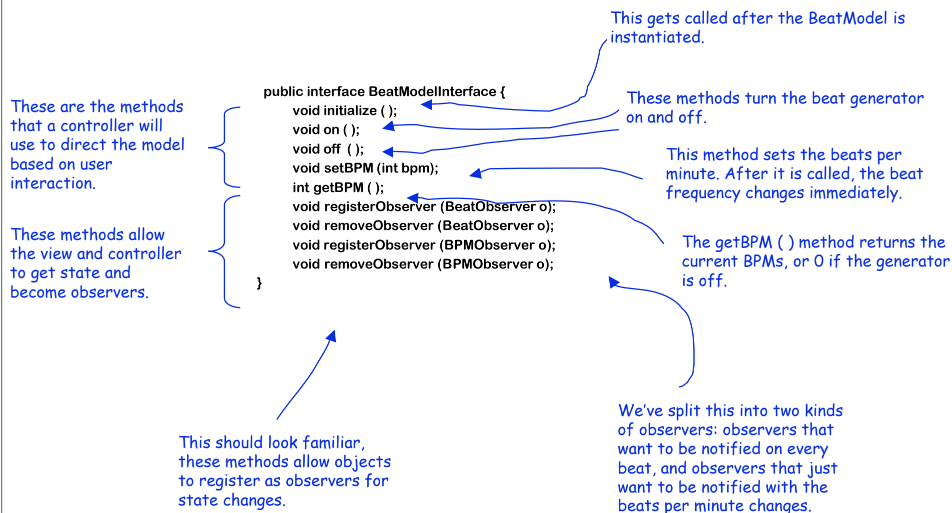
These are the methods that a controller will use to direct the model based on user interaction.

These methods allow the view and controller to get state and become observers.

These methods turn the beat generator on and off.

This method sets the beats per minute. After it is called, the beat frequency changes immediately.

The getBPM ( ) method returns the current BPMs, or 0 if the generator is off.



```
public interface BeatModelInterface {
    void initialize ( );
    void on ( );
    void off  ( );
    void setBPM (int bpm);
    int getBPM ( );
    void registerObserver (BeatObserver o);
    void removeObserver (BeatObserver o);
    void registerObserver (BPMObserver o);
    void removeObserver (BPMObserver o);
}
```

This should look familiar, these methods allow objects to register as observers for state changes.

We've split this into two kinds of observers: observers that want to be notified on every beat, and observers that just want to be notified with the beats per minute changes.

21

Now lets take a look at the concrete BeatModel class….

We implement the BeatModelInterface.

This is needed for the MIDI code.

```
public class BeatModel implements BeatModelInterface, MetaEventListener {
    Sequencer sequencer;
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int bpm = 90;
    // other instance variables

    public void initialize() {
        setUpMidi();
        buildTrackAndStart();
    }
    public void on() {
        sequencer.start();
        setBPM(90);
    }
    public void off() {
    setBPM(0);
      sequencer.stop();
    }
    public void setBPM (int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(getBPM());
        notifyBPMObservers();
    }
    public int getBPM() {
    return bpm;
    }
    void beatEvent ( ) {
        notifyBeatObservers ( );
    }
    // code to register and notify observers
    // Lots of MIDI code to handle the beat.
}
```

The sequencer is the object that knows how to generate the real beats (that you can hear!)

The ArrayLists hold the two kinds of Observers

bpm holds the frequency of beats. Default = 90.

This method does setup for the sequencer and sets up the beat tracks for us.

The on() and off ( ) starts and shuts off the sequencer.

(1) Sets the bpm instance variable
(2) Asks the sequencer to change its beats
(3) Notifies all BPM observers that the BPM has changed.

The beatEvent ( ) method, which is not in the BeatModelInterface, is called by the MIDI code whenever a new beat starts. This notifies all BeatObservers that a new beat has just occurred.

Implementing the View (just an outline!)

DJView is an observer for both the real-time beats and BPM changes.

```
public class DJView implements ActionListener, BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;

    public DJView ( ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver ( (BeatObserver) this);
        model.registerObserver ((BPMObserver)this);
    }

    public void createView ( ){
        // create all Swing components here
    }
    public void updateBPM ( ) {
        int bpm = model.getBPM ( );
        if (bpm == 0) {
            bpmOutputLabel.setText ("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM ( ));
        }
    }
    public void updateBeat ( ) {
        beatBar.setValue (100);
    }
}
```

The view holds a reference to both the model and the controller.

The updateBPM ( ) method is called when a state change occurs in the model. When that happens we update the display with the current BPM. We can get this value by requesting it directly from the model.

Likewise, the updateBeat ( ) method is called when the model starts a new beat. When that happens, we need to pulse our "beatbar". We do this by setting it to its maximum value and letting it handle the animation of the pulse.

The DJView continued… (the user interface control)

```
public class DJView implement ActionListener, BeatObserver, BPMObserver {
        BeatModelInterface model;
        ControllerInterface controller;
        Jlabel bpmLabel;
        JTextField bpmTextField;
        Jbutton setBPMButton;
        Jbutton increaseBPMButton;
        Jbutton decreaseBPMButton;
        JMenuBar menuBar;
        Jmenu menu;
        JMenuItem startMenuItem;
        JMenuItem stopMenuItem;

        public void createControls ( ) {
                // create all Swing components
        }
        public void enableStopMenuItem  ( ){
                stopMenuItem.setEnabled(true);
        }
        public void disableStopMenuItem ( ){
                stopMenuItem.setDisabled (true);
        }
        // same for startMenuItem
        public void actionPerformed (ActionEvent event) {
                if (event.getSource ( ) == setBPMButton) {
                        int bpm = Integer.parseInt (bpmTextField.getText ( ) );
                        controller.setBPM (bpm);
                } else if (event.getSource( ) == increaseBPMButton) {
                        controller.increaseBPM () ;
                } else if (event.getSource( ) == decreaseBPMButton) {
                        controller.decreaseBPM () ;
                }
        }
}
```

This method is called when a button is clicked.

If the Set button is clicked then it is passed on to the controller along with the new bpm.

Likewise if increase or decrease buttons are clicked, this information is passed on to the controller.

24

Now for the Controller….

- Remember - the controller is the strategy that we plug into the view.
- What does a Strategy pattern look like? What do we need?

```
public interface ControllerInterface {
                void start ( );
                void stop ( );
                void increaseBPM ( );
                void decreaseBPM ( );
                void setBPM (int bpm);
}
```

Here are all the methods that a view can call on the controller

These should all look familiar after seeing the model's interface. You can start and stop the beat generation, and change the BPM. This interface is "richer" than the BeatModel interface because you can adjust the BPMs with increase and decrease.

25

The Controller

```
public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }
    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }
    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }
    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }
    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm - 1);
    }
    public void setBPM (int bpm) {
        model.setBPM (bpm);
    }
}
```

Controller implements the ControllerInterface!

The controller is the creamy stuff in the middle of the MVC oreo cookie, so it is the object that gets to hold on to the view and the model and glues it all together.

The controller is passed the model in the constructor and then creates the view.

When you choose Start from the user interface menu, the controller turns the model on and then alters the user interface to that the start menu item is disabled and the stop menu item is enabled.

Note: the controller is making the intelligent decision for the view. The view just knows how to turn menu items on and off; it doesn't know the situations in which it should disable or enable them.

Putting it all together…

```
public class DJTestDrive {
    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel ( );
        ControllerInterface controller = new BeatController (model);
    }
}
```

First create the model…

…..then create a controller and pass it the model. Remember, the controller creates the view, so we don't have to do that.

Summary

- The MVC Pattern is a compound pattern consisting of the Observer, Strategy and Composite patterns.
- The model makes use of the Observer pattern so that it can keep observers updated, yet stay decoupled from them.
- The controller is the strategy for the view. The view can use different implementations of the controller to get different behavior.
- The view uses Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames, and buttons.
- These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible.
- The Adapter pattern can be used to adapt a new model to an existing view and controller.

28